How GLM5.3 Sparse Attention Affects HBM Memory Usage

GLM-5.3, KV Cache Offloading, HiSparse, AgentX TileRT, InferenceX DeepSeek Sparse Attention, IndexShare, Single-rollout Asynchronous Optimization, Cybersecurity

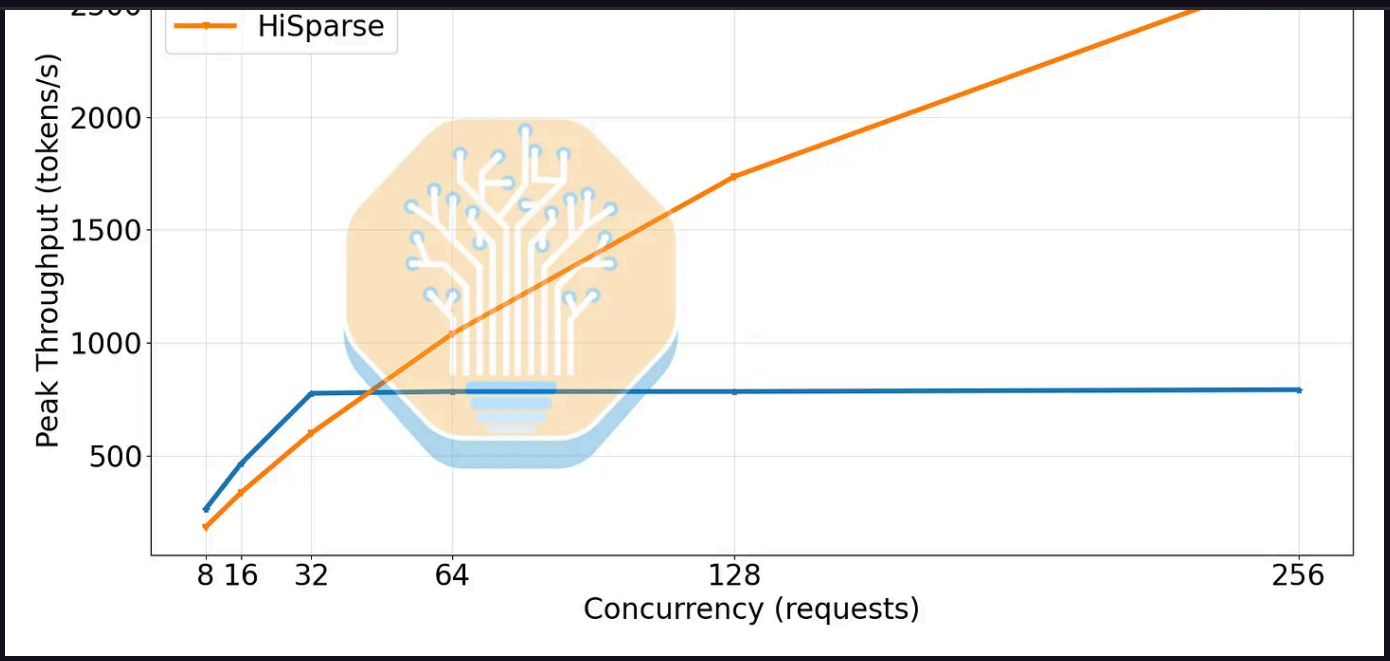
KIMBO CHEN, ALEC IBARRA, WENYAO GAO, AND 3 OTHERS

44 MINUTES AGO • PAID

4 Share

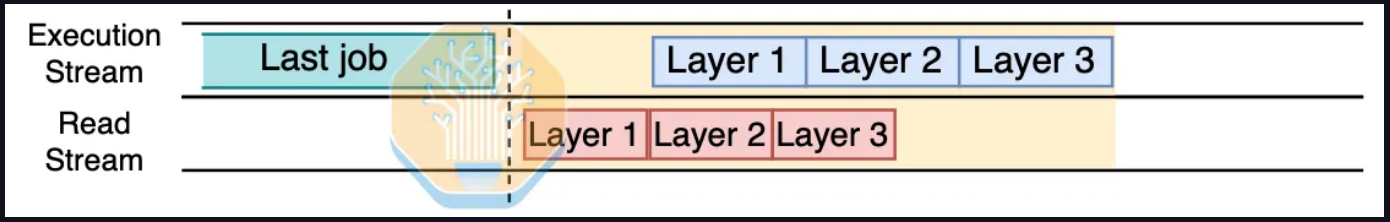
How Sparse Attention Affects DRAM/NAND Memory

How does sparse attention affect the TAM of memory, including HBM and NAND? Sparse attention selects top-k most relevant tokens to attend to, reducing the memory consumption and bandwidth requirements during the core Scaled Dot-Production Attention (SDPA) operation. However, **the efficiency improvement doesn't directly translate to overall memory savings in practice.** Concretely, the top-k selection operation typically requires the full context to be in HBM, so **sparse attention doesn't eliminate the memory capacity bottleneck.**



Sparse attention throughput is bottlenecked by memory capacity. Source: HiSparse

To overcome this limitation, the SGLang team designed HiSparse, a hierarchical memory system that proactively offloads KV cache entries from device HBM to host DRAM. **HiSparse behaves like an LRU (Least Recently Used) cache**, where it loads tokens from DRAM to HBM upon top-k selection cache miss, and it offloads tokens from HBM to DRAM based on the LRU eviction policy. To reduce the cache miss latency, HiSparse overlaps KV cache loading of layer N with the execution of layer N-1 (layer-wise overlapping), introduced in HiSparse's prior work HiCache.



Layer-wise overlapping. Source: CachedAttention

With HiSparse, SGLang greatly boosts throughput at high concurrency and long context scenarios, at the cost of top-k cache miss I/O overhead.

Sparse attention reduces KV cache memory and bandwidth requirements at the SDPA operation, but it does not reduce the overall memory capacity usage. In addition, HiSparse shows that system optimizations can overcome sparse attention's memory capacity

attention models, here we explain the design of Z.ai's GLM-5 model series, and how they are served in practice.

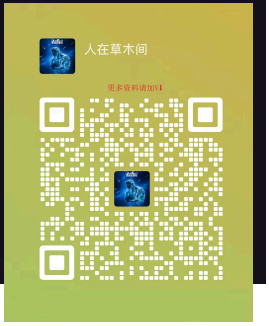
GLM5.3 Agentic Inference Serving

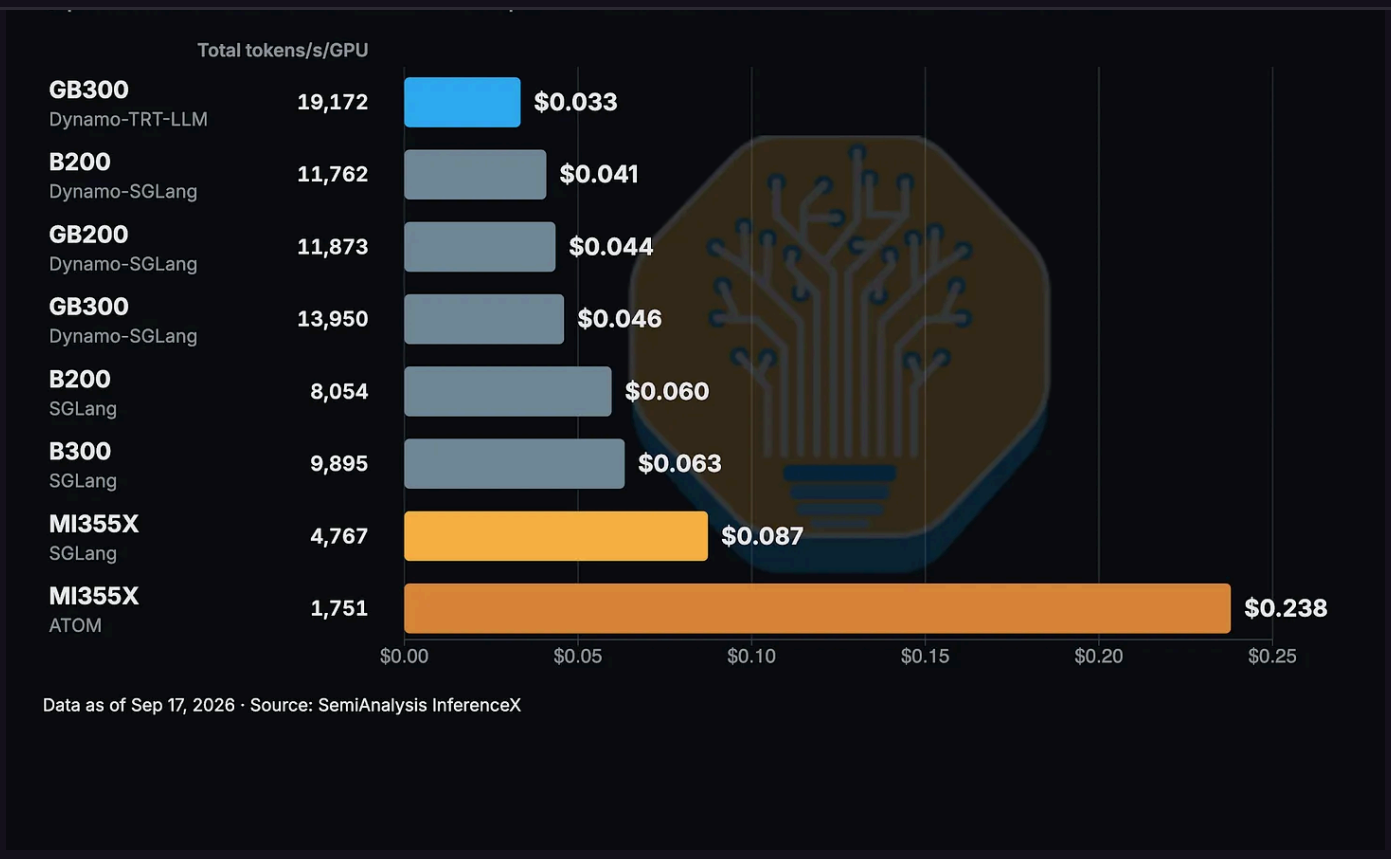
In our realtime benchmark board InferenceX, GB300 delivers the lowest modeled serving cost in our comparison at a response speed of 150 tokens per second. GB300 handles more tokens per GPU, and its higher hourly cost still leaves it ahead of GB200 on cost efficiency. The GB300 result uses Dynamo-TRT-LLM, while the GB200 result uses Dynamo-SGLang.

We use the September 17 AgentX snapshot results to help illustrate the serving implications of GLM-5.3. We estimate costs using InferenceX's model of owning and operating infrastructure at hyperscaler scale. Total tokens include input and generated output, including cached input history reused across turns. Streaming speed is measured using p90 interactivity; time to first token is evaluated separately.

At 150 tokens per second, GB200 costs approximately $0.044 per million total tokens, compared with $0.238 for MI355X running ATOM. That is about 82% lower cost at the same streaming-speed target. These values are estimated from the measured curves, rather than from a separate test at exactly 150 tokens per second.

For this comparison, we use InferenceX's estimated cost of owning and operating the infrastructure at hyperscaler scale. Total tokens include input and generated output, including input history reused from cache.





Source: SemiAnalysis InferenceX current InferenceX dashboard; interpolation method; GB300 source run; MI355X source run.

Among the Dynamo-SGLang results, GB300 serves roughly 13950 total tokens per second per GPU here, a 17.5% throughput advantage versus 11873 total tokens for GB200. However, we assume $2.31 per GB300 GPU hour versus $1.86 for GB200. The higher hourly cost more than offsets GB300's throughput lead at this target.

In the comparison with AMD, GB200 costs about 52% less than ATOM at 100 tokens per second, 62% less at 125, and 82% less at 150. Across these three targets, the gap widens as response speed increases.

Counting only generated output, GB200 costs $5.92 per million output tokens at a response speed of 150 tokens per second, which is 79% less than $28.52 for MI355X ATOM. This metric is useful to analyze cost under agentic workloads, where agents repeatedly reuse long input histories while generating relatively little new text.

Source: SemiAnalysis InferenceX; current InferenceX dashboard

The GB200 measurements on either side of the 150 token per second target have p90 TTFT (time to first token) of 14-19 seconds, compared with 1.3-1.7 seconds for MI355X ATOM. Some of the GB200 lowest cost runs have much longer waits for the first token.

We therefore make a second comparison using only configurations that were actually tested and achieved at least 150 tokens per second with p90 TTFT below two seconds. The chart below shows B200 with Dynamo-SGLang at $0.0666 per million total tokens versus $0.1265 for MI355X with SGLang. B200 still costs about 47% less with the first-token limit.

Source: SemiAnalysis InferenceX B200 run 442071; GB300 run 441597; MI355X run 442075

Optimizations

In GLM's 5.2 / 5.3 model, its attention architecture reduces memory demands, and long running agents still need to retain earlier conversation history. GLM reduces the cost of handling this history in two ways: KV compression reduces the cached state stored per token, and sparse attention reduces how much of that state each attention operation reads. Then the serving engine would determine where to store the cache and how to retrieve it efficiently.

These B200 results show how much reuse can happen outside of GPU memory. When concurrency increases from 8 to 16 requests, the share of prompt tokens reused from GPU memory falls from 90.3% to 54.8%. Much of that reuse shifts to host memory, which rose from 6.0% to 40.3%. Much of the decline in GPU cache is offset by reuse from host memory, keeping the overall cache hit rate above 95% at all concurrency levels.

Source: SemiAnalysis InferenceX; InferenceX B200 rows 440962, 440961, and 440959; workflow run 33683520699

Beyond cache management, changes to how the serving engine executes prefill and decoding can also change performance. Two separate GLM-5.2 studies illustrate this:

- vLLM keeping the first decoding step on a consistent CUDA graph execution path reduced average TPOT (time per output token) from about 40 ms to 22 ms in an NVFP4 study.
- On a different 8 MI355X long context workload, ATOM engine processing prefill in chunks across pipeline stages delivered 98% higher total throughput and reduced median TTFT from 28.6 to 8.7 seconds.

TileRT

overhead and overlapping computation, memory access, and communication.

It prioritizes faster token generation per user, rather than maximum batched throughput, while vLLM handle prefill.



Ultra-High Interactivity on NVIDIA GPUs? - TileRT InferenceX

BRYAN SHAN, DANIEL NISHBALL, AND 4 OTHERS · 8月10日

Read full story →

On AgentX, FP8 TileRT MI355X achieves 2x the P90 interactivity than the best FP4 MI335X config. Comparing to GB300 NVL72, it brings a 40% uplift in interactivity. These are serving metrics that used to require specialized hardware.

DeepSeek Sparse Attention

GLM-5 (and GLM-5.x models) is a 744B total, 40B active-parameter mixture-of-experts model. For every token, it has 1 shared expert and is routed through 8 out of 256 experts, which is sparsity 32. It features DeepSeek Sparse Attention, which we will discuss in this section.

DeepSeek Sparse Attention (DSA), introduced in DeepSeek V3.2, consists of two components: A **lightning indexer** that selects top K tokens, and a **sparse Multi-Latent Attention** (MLA).

Lightning Indexer

A lightning indexer is functionally similar to a **lightweight attention mechanism**. Queries and keys are down-projected to lower dimensions, with indexer queries being multi-headed and indexer keys single-headed. Since we only need a query and key relation score (logits) for selecting top K tokens, the indexer doesn't need value embeddings or the softmax operation for normalized logits. Instead, it computes a query-key dot product, followed by a ReLU non-linearity. The indexer then computes each token's indexer score as a weighted sum of logits across all query heads, which we use to select the top K tokens with. This means despite the indexer having multiple heads, **token selection is shared across multiple attention heads**. When selecting top K tokens, we keep **queries that attend to less than K tokens as dense attention**, and we select top K otherwise.

Conceptual example of the lightning indexer. Source: SemiAnalysis

Indexer score is a weighted sum of all indexer query heads. Source: SemiAnalysis

Sparse MLA

As we explained in our Kimi K3 article, MLA operates in two modes: Multi-Head Attention (MHA) mode and Multi-Query Attention (MQA) mode. Both come with trade-offs: MHA mode has lower FLOPs but 42x higher memory cost, MQA mode has lower memory cost but up to 3.4x higher FLOPs. DSA uses MQA mode, since sparse attention attends to less tokens,mitigating the higher FLOP usage. However, in practice, there is a sequence length threshold where under it, MHA mode is actually more efficient that MQA mode. The intuition is that at shorter sequence length, the memory loading time isn't as dominating,

about 77K for tensor parallel. The 2K minimum is due to top K being 2048, and as mentioned above, attention below 2K is dense.

MHA (green) can be faster than MQA (red). Source: vLLM PR #48770

MLA Modifications

Comparing the MLA dimension configurations of GLM-5 and DeepSeek V3.2, we see that the most notable differences are the **number of query heads** and the **query key head dimension**.

GLM-5 tech report mentions DeepSeek chose the number of query heads according to the roofline of H800. This is likely referring to the fact that **MLA's SDPA arithmetic intensity**

- `d` : Head dimension
- `r` : RoPE dimension
- `H` : Number of heads
- `b` : Number of bytes per parameter

In MQA mode, the FLOPs are

```
2 * H * L * (d+r)  # P = Q [H, d+r] @ K.T [d+r, L]
2 * H * L * d      # O = P [H, L] @ V [L * d]
```

and the memory loaded is

```
b * H * (d+r)  # Q
b * L * (d+r)  # K and V
```

So the arithmetic intensity is

```
(2 * H * L * (2 * d + r)) / (b * (d+r) * (L + H))
= (2 * H * L * (2 * d + r)) / (b * L * (d + r))  # L >> H
= (2 * H * (2 * d + r)) / (b * (d + r))
```

If we plug in the DeepSeek V3.2 configuration `d` = 512, `r` = 64, `b` = 2), we get

```
(2 * H * (2 * 512 + 64) / (2 * (512 + 64)) ~= 2 * H
```

H800's practical arithmetic intensity is 258.2 FLOP/B (Practical peak 865 TFLOP/s, 3.35 TB/s bandwidth, according to DeepSeek), and plugging in the formula gives us `H` ~= 128.

The fact that GLM-5 has H = 64, half the number of query heads, implies that GLM-5 may be designed for different hardware. If we plug in GLM-5's configuration (d = 512, r = 64, b = 2, H = 64), we get an arithmetic intensity of 120.8 FLOP/B. Thus, we suspect GLM-5 is optimized for Moore Threads MTT S4000, which is about 128 FLOP/B. Moore Threads' collaboration with Z.ai, such as GLM-5.3-Flash day 0 support, corroborates our theory.

the SDPA head dimension from 192 to 256, which is still a 33% FLOP reduction after considering the head number reduction.

IndexShare

The latency cost of the indexer becomes non-negligible as context grows. To mitigate the cost, Z.ai proposes IndexShare (aka IndexCache) in GLM-5.2, where every 4 DSA layers share one indexer.

Latency cost at different context lengths for a 30B DSA model. Source: IndexCache

IndexShare introduces complications in training and inference. Standard DSA training involves two stages: A dense warm-up stage where all model weights except the indexer are frozen and attention computation is dense, and then a sparse adaptation stage where the full model is trained with sparse attention.

In terms of training objective, the indexer is trained to align with the sum of attention head scores, using a KL divergence objective. IndexShare adapts the training objective to **aligning the indexer logit distribution with the average distribution of attention scores across shared layers**.

For inference, typically **the indexer caches keys, like attention caches KV.** Since multiple layers share one indexer, we need to additionally cache the top K indices at the full layer, in order to reuse the top K selection information at shared layers.

This design reduces indexer cache by 75%, reduces indexer FLOPs by 75%, and boosts throughput by 1.5x to 1.8x at different context lengths and prefill/decode.

Baseline vs. GLM-5.2 configuration (red) for a 30B DSA model. Source: IndexCache

Post-Training Design

Pipeline

After pre-training and mid-training, GLM-5's post-training pipeline starts with supervised fine-tuning (SFT), continues with **three RL training stages** (reasoning RL, agentic RL, and general RL), and ends with an **on-policy cross-stage distillation**, distilling reasoning and general RL back into the SFT checkpoint.

GLM-5 post-training pipeline. Source: SemiAnalysis

Here we highlight two details. First, the tech report states that the reasoning RL model is trained **entirely on-policy**, implying that it is trained synchronously. Synchronous RL training is low system efficiency but has higher RL training stability. Second, **on-policy cross-stage distillation is similar to Multi-teacher On-Policy Distillation** (MOPD), where a student model is RL trained with on-policy distillation over multiple teacher models. However, unlike MOPD, whose goal is to merge expert teacher models into one model, GLM-5 here seems to be more about **recovering old capability loss** when the model learns new capabilities. Following this assumption, we are not sure why the Z.ai only uses reasoning and general RL checkpoints as teachers, instead of using all three RL checkpoints.

Based on the announcement blog post, GLM-5.2 may have replaced the on-policy cross-stage distillation with a standard MOPD pipeline, which Z.ai named parallel OPD training. Z.ai touted its scale and training efficiency: they merged more than ten expert models into a final model by training for about two days.

RL Training Algorithm

In the synchronous RL stage, Z.ai adopts Group Relative Policy Optimization (GRPO)-based RL algorithm with modern adaptations, including:

- **Remove KL regularization term** to improve system efficiency and allow more aggressive gradient updates
- **IcePop**: Apply token-level masking based on the training-inference mismatch ratio
- **Clip-Higher**: Use a higher max threshold for importance sampling ratio to encourage exploration

In contrast, Z.ai employs a REINFORCE-like RL algorithm for asynchronous RL stage, adding Direct Double-Sided Importance Sampling (DIS), an IcePop-like clipping mechanism. Below we compare the token-level objectives for the two RL stages:

Token-level objective for the two RL stages. Source: SemiAnalysis

The importance sampling (IS) ratio compares **the probability of each sampled action under the current policy and the behavior policy**, scaling each action's contribution to the training objective. For IS ratio, the synchronous RL stage utilizes the **trainer policy of the previous iteration** for the behavior policy, whereas the asynchronous RL stage utilizes the **inference policy** for that. In an asynchronous training setting, policy updates happen multiple times during rollouts, so we will need every checkpoint version in between to calculate the IS ratio. However, this is infeasible due to the high memory and latency overhead. As a result, Z.ai opts for using inference policy for efficiency and simplicity, at the cost of numeric stability challenges.

IS ratio for the two RL stages. Source: SemiAnalysis

Finally, we highlight the major differences in advantage calculation in RL training, cross-distillation, and long-horizon RL training. RL training employs the GRPO standard group-normalized advantage, and cross-distillation uses the reverse KL divergence.

Advantage calculation comparison. Source: SemiAnalysis

Single-Rollout Asynchronous Optimization

Long horizon tasks produce extremely long trajectories, and in the context of GRPO, the high variance in trajectory length produces unstable training signals that may skew towards longer trajectories. What's worse, since RL rollout workload is sensitive to end-to-end latency, the tail-latency of a straggler rollout worsens for long horizon tasks. To mitigate this issue, Z.ai introduced Single-rollout Asynchronous Optimization (SAO) for long-horizon RL training in GLM-5.2. **SAO replaces group-normalized advantage with a modified Generalized Advantage Estimation** (GAE), inspired by Proximal Policy Optimization (PPO). GAE only needs a single rollout to compute the advantage. This removes the need to do multiple rollouts per prompt, mitigating GRPO's short-come.